ANÁLISIS DEL USO DE SPRING BOOT EN EL DESARROLLO DE MICROSERVICIOS Y SU IMPACTO EN LA CALIDAD DEL SOFTWARE

CARLOS MARIO FONTALVO GARCÍA

INSTITUCIÓN UNIVERSITARIA PASCUAL BRAVO FACULTAD DE INGENIERÍA

TECNOLOGÍA EN DESARROLLO DE SOFTWARE

SANTA MARTA

2026

ANÁLISIS DEL USO DE SPRING BOOT EN EL DESARROLLO DE MICROSERVICIOS Y SU IMPACTO EN LA CALIDAD DEL SOFTWARE

Carlos Mario Fontalvo García

Trabajo de grado para optar al título de Tecnólogo en Desarrollo de Software

Asesor
Liliana María García Aguirre
Magister en Ingeniería de Software

INSTITUCIÓN UNIVERSITARIA PASCUAL BRAVO
FACULTAD DE INGENIERÍA
TECNOLOGÍA SISTEMAS MECATRÓNICOS
SANTA MARTA
2026

Contenido

Lista de figuras 5

Lista de tablas 6

Resumen 7

Abstract 8

Glosario 9

Introducción 10

1. Planteamiento del problema 11

1.1 Descripción 11

1.2 Formulación 12

2. Justificación 13

3. Objetivos 14

3.1 Objetivo general 14

3.2 Objetivos específicos 14

4. Marco Teórico 15

5. Metodología 16

5.1 Tipo de proyecto 16

5.2 Método 16

5. 3 Instrumentos de recolección de información 17

5.3.1 Fuentes primarias. 17

5.3.2 Fuentes secundarias. 18

6. Resultados 19

6.1 Evolución del desarrollo backend hacia arquitecturas de microservicios 19

6.2 Comparación de Spring Boot con otras tecnologías para el desarrollo de microservicios 21

6.2.1 Desarrollo backend para aplicaciones web, Servicios Web Restful: Node.js vs Spring Boot 21

6.2.2 .NET Core Vs. Java Spring Boot: A Review-Driven Performance Assessment for Cloud-Native API Architectures 22

6.2.3 Performance Analysis of Enterprise Frameworks: Spring Boot and Play Framework 24

6.2.4 Comparative Analysis of Java Frameworks for Microservices: Spring Boot, Micronaut, and Quarkus 25

6.3 Spring Boot: Características y ventajas en arquitecturas de microservicios 29

6.3.1 ¿Qué es Spring Boot? 29

6.3.2 IDEs y herramientas mínimas requeridas 29

6.3.3 ¿Qué son las dependencias y por qué son importantes? 30

6.3.4. Arquitecturas que se pueden utilizar en Spring Boot 30

6.3.5 ¿Qué función cumplen las anotaciones? 31

6.3.6 Pasos para crear un proyecto en Spring Boot con Microservicios 32

7. Conclusiones 40

8. Recomendaciones 42

9. Referencias bibliográficas 43

Lista de figuras

Figura 1. Interfaz de configuración en Spring Initializr 33
Figura 2. Organización en capas típicas para microservicios 34
Figura 3. *C*onfiguración del archivo application.properties para el microservicio 35
Figura 4. Instrucciones de Dockerfile para la contenedorización de la aplicación 36
Figura 5. Sectores y organizaciones líderes que implementan Spring Boot en Colombia y Latam 37
Figura 6. Ingenieros por framework en Colombia (2023) 38
Figura 7. Impacto de las buenas prácticas en el costo del desarrollo de software 39

Lista de Tablas

Tabla 1. Comparativo por características Spring Boot, Micronaut y Quarkus 26

Tabla 2. Arquitecturas soportadas en Spring Boot..... 30

Tabla 3. Anotaciones esenciales en Spring Boot 32

Resumen

ANÁLISIS DEL USO DE SPRING BOOT EN EL DESARROLLO DE MICROSERVICIOS Y SU IMPACTO EN LA CALIDAD DEL SOFTWARE

CARLOS MARIO FONTALVO GRACIA

En este trabajo de grado se aborda el análisis del ecosistema de Spring Boot aplicado al desarrollo de soluciones empresariales basadas en arquitecturas distribuidas de microservicios, centrándose en cómo una implementación deficiente puede acarrear problemas críticos de acoplamiento y aumentar la deuda técnica del sistema.

La investigación se realizó ante la necesidad de establecer criterios arquitectónicos claros en las organizaciones para mitigar los altos costos operativos, fallas de mantenimiento y pérdidas financieras que producen los sistemas improvisados en el sector tecnológico regional. Para resolver este planteamiento, se empleó un método integral con enfoques descriptivos y explicativos, sustentado en la revisión de estándares de la industria.

Como principal hallazgo, se menciona que el framework optimiza la calidad global del software si su autoconfiguración se articula con patrones avanzados de resiliencia, modularidad y diseño guiado por el dominio, logrando reducir drásticamente el retrabajo en producción. Lo más relevante del cumplimiento de los objetivos fue demostrar que la estandarización técnica ejerce una influencia directa en la viabilidad económica y la escalabilidad del sistema, concluyendo que las buenas prácticas no deben percibirse como un sobrecosto inicial sino como una ventaja competitiva.

Abstract

This undergraduate thesis addresses the analysis of the Spring Boot ecosystem applied to the development of enterprise solutions based on distributed microservices architectures, focusing on how inadequate implementation practices can lead to critical coupling issues and increase the system's technical debt. The research was conducted in response to the need for establishing clear architectural criteria within organizations to mitigate the high operational costs, maintenance failures, and financial losses caused by improvised software systems in the regional technology sector. To address this problem, a comprehensive methodology was employed, combining descriptive and explanatory approaches supported by a review of industry standards.

As a primary finding, the study highlights that the framework optimizes overall software quality when its auto-configuration capabilities are integrated with advanced resilience, modularity, and domain-driven design patterns, significantly reducing production rework. The most relevant outcome in achieving the research objectives was demonstrating that technical standardization has a direct impact on both the economic viability and scalability of software systems. The study concludes that best practices should not be perceived as an initial overhead cost, but rather as a competitive advantage.

Glosario

Acoplamiento: Grado de interdependencia entre los módulos o componentes de un sistema informático. Un acoplamiento bajo es deseable para facilitar el mantenimiento y la escalabilidad.

API Gateway: Componente de infraestructura que actúa como punto de entrada único para gestionar, enrutar y proteger las peticiones dirigidas a un ecosistema de microservicios.

Bulkhead: Patrón de aislamiento de recursos que divide el sistema en compartimentos para evitar que una falla en un componente específico agote los recursos de todo el sistema.

Circuit Breaker: Patrón de diseño que interrumpe las llamadas a un servicio remoto cuando se detecta un número elevado de fallos, evitando la saturación del sistema y permitiendo su recuperación.

DevOps: Cultura y metodología de trabajo que busca la colaboración estrecha entre los equipos de desarrollo y operaciones para automatizar y agilizar el ciclo de vida del software.

Domain-Driven Design (DDD): Enfoque de diseño de software en el que la estructura y el lenguaje del código se alinean directamente con las reglas y procesos del dominio del negocio.

End-to-End (E2E): Metodología de pruebas que valida el funcionamiento de un sistema completo, desde el flujo inicial hasta el resultado final, simulando el comportamiento real del usuario.

Integración y Despliegue Continuo (CI/CD): Estrategia de automatización que permite integrar cambios de código de forma frecuente y liberarlos en entornos de producción de manera ágil y segura.

JWT (JSON Web Token): Estándar compacto para la transmisión segura de información entre partes en forma de objeto JSON, utilizado principalmente para la autenticación en aplicaciones web.

Mecanismos de Fallback: Respuestas alternativas predefinidas que el sistema ejecuta cuando un servicio principal falla, garantizando que el usuario reciba una respuesta degradada pero funcional.

OAuth: Protocolo estándar de autorización que permite a las aplicaciones acceder a recursos de usuario en otros sitios sin necesidad de compartir las contraseñas.

Retries: Lógica de reintentos automáticos aplicada a operaciones que fallan debido a errores transitorios, con el fin de aumentar la resiliencia del sistema.

Timeouts: Configuración de tiempo límite de espera para una respuesta externa, diseñada para evitar que el sistema quede bloqueado indefinidamente ante un servicio lento o caído.

Zero Trust: Marco de seguridad basado en la premisa de no confiar de forma predeterminada en ningún usuario o dispositivo, exigiendo verificación continua para cada solicitud de acceso.

Introducción

El proyecto contiene un análisis detallado sobre el uso de Spring Boot en el desarrollo de arquitecturas basadas en microservicios y cómo esta herramienta impacta la calidad del software, el tema central aborda la transición desde modelos monolíticos tradicionales hacia sistemas distribuidos, donde se aborda la necesidad actual de las organizaciones de contar con aplicaciones flexibles, escalables y de alta disponibilidad que respondan a la competitividad del mercado tecnológico actual.

La investigación responde a la necesidad crítica de establecer mejores prácticas de arquitectura, ya que un desarrollo improvisado o sin rigor técnico genera "deuda técnica" que, en el futuro, deriva en graves problemas de mantenimiento y escalabilidad. Estas fallas no solo influyen negativamente en los costos operativos, sino que pueden comprometer la estabilidad del negocio, llevando incluso al colapso empresarial ante la imposibilidad de adaptar o sostener sus sistemas tecnológicos en un mercado competitivo.

El trabajo se estructura de una manera secuencial para abordar de una manera específica la temática a tratar donde en primera instancia se presenta el planteamiento del problema y una justificación que contextualiza la relevancia de los microservicios a nivel global y regional, continuando con el desarrollo de un marco teórico que sustenta los principios de diseño y patrones arquitectónicos esenciales, para luego exponer una sección de resultados con comparativas técnicas entre Spring Boot y otras tecnologías como Node.js, .NET Core y Play Framework, finalizando con la propuesta de lineamientos prácticos y un prototipo funcional que valida la aplicabilidad de las recomendaciones sugeridas.

El método empleado para este estudio es una estrategia metodológica integral que combina enfoques descriptivos y explicativos. El proceso comenzó de forma inductiva mediante la revisión de fuentes primarias y secundarias, como documentación oficial y artículos científicos, para identificar las mejores prácticas de la industria. Luego, se aplicó un razonamiento deductivo para plantear hipótesis sobre el impacto de estas prácticas en atributos como la mantenibilidad y la seguridad. Finalmente, se utilizó la observación técnica y el análisis comparativo de proyectos para verificar empíricamente los beneficios del uso sistemático de Spring Boot frente a enfoques no estructurados.

En cuanto a las limitaciones del trabajo, se centran específicamente en el ecosistema de Java y Spring Boot, por lo que los hallazgos pueden no ser directamente transferibles a otros lenguajes de programación o entornos de desarrollo con lógicas distintas. Además, aunque se evalúan métricas de desempeño y calidad, estas se encuentran sujetas a escenarios de prueba controlados y prototipos funcionales, lo que implica que en entornos de producción masiva con variables externas impredecibles, los resultados podrían presentar matices adicionales no cubiertos exhaustivamente en este informe.

1. Planteamiento del problema

1.1 Descripción

En la actualidad, el desarrollo de aplicaciones empresariales demanda cada vez más la adopción de arquitecturas orientadas a microservicios, este enfoque consiste en fragmentar una aplicación en pequeños servicios independientes que se comunican entre sí a través de una red, donde cada servicio es responsable de una funcionalidad específica y puede desarrollarse, implementarse y escalarse de manera autónoma.

En contraste, la arquitectura monolítica se caracteriza por poseer una única base de código en la que las funcionalidades están estrechamente acopladas, lo que facilita el desarrollo inicial, pero presenta limitaciones en términos de escalabilidad, mantenibilidad y despliegue independiente de las actualizaciones

En este contexto, Java proporciona un ecosistema sólido de herramientas que permiten optimizar el desarrollo de aplicaciones basadas en microservicios. Entre ellas destaca Spring Boot, este framework se ha consolidado como una de las principales opciones para la construcción de microservicios debido a su capacidad para simplificar el ciclo de desarrollo mediante la integración de configuraciones automáticas y librerías preconfiguradas.

Sin embargo, el uso de Spring Boot no garantiza por sí mismo la calidad del software desarrollado. La ausencia de criterios claros durante su implementación puede derivar en sistemas con bajo rendimiento, dificultades de mantenimiento y escasa escalabilidad, generando arquitecturas que, aunque distribuidas, conservan problemas propios de sistemas monolíticos. Esta situación evidencia la necesidad de analizar cómo el uso de Spring Boot influye realmente en la calidad del software cuando se adopta una arquitectura de microservicios.

Diversos enfoques de desarrollo, como la aplicación de principios de diseño, la gestión adecuada de la configuración, la documentación de APIs y la implementación de pruebas automatizadas, han sido asociados con mejoras en la mantenibilidad y reducción de errores en producción. No obstante, en el contexto de proyectos de desarrollo controlados, aún resulta necesario evaluar el impacto que tiene el uso de Spring Boot en la calidad del software, más allá de su adopción como herramienta tecnológica.

Esta necesidad se refuerza por el crecimiento sostenido del mercado de orquestación de microservicios, el cual, según Nester Research (2025), mantiene una tendencia de expansión impulsada por la adopción de arquitecturas modulares y prácticas orientadas a la eficiencia del desarrollo. Ante este panorama, se hace relevante analizar de manera sistemática el impacto del uso de Spring Boot en el desarrollo de microservicios, con el fin de contribuir a la comprensión de su influencia en la calidad del software y apoyar la toma de decisiones en entornos de desarrollo.

1.2 Formulación

¿Cómo influye el uso de Spring Boot en el desarrollo de arquitecturas de microservicios y cuál es su impacto en la calidad del software?

2. Justificación

La importancia de analizar el uso de Spring Boot en el desarrollo de microservicios y su impacto en la calidad del software radica en la necesidad de fortalecer la implementación de buenas prácticas durante el diseño y construcción de soluciones empresariales. En un entorno tecnológico altamente competitivo, las organizaciones demandan aplicaciones flexibles, escalables y fácilmente modificables que respondan con agilidad a los cambios del mercado. La arquitectura de microservicios, especialmente cuando se implementa con Spring Boot, se ha consolidado como una tendencia dominante en el desarrollo de aplicaciones empresariales modernas. Según el informe de Fortune Business Insights (2025), el mercado global de microservicios en la nube fue valorado en USD 1.84 mil millones en 2024 y se proyecta que alcanzará los USD 8.06 mil millones en 2032, con una tasa de crecimiento anual compuesta del 20.3%, lo que evidencia el creciente protagonismo de estos modelos en sectores como la banca, la salud y el comercio electrónico.

No obstante, en Latinoamérica se evidencian dificultades asociadas a la inadecuada aplicación de principios técnicos y estándares de calidad, lo que genera altos costos de mantenimiento, vulnerabilidades, baja escalabilidad y pérdida de productividad en los equipos de desarrollo. De acuerdo con un estudio de Tricentis (2025), el 76% de las organizaciones en la región implementa cambios de código sin realizar pruebas completas, y en Colombia el 42% de las empresas puede perder hasta 5 millones de dólares anuales debido a fallas de software. Esta problemática no es exclusiva de la región. Según el Consortium for Information & Software Quality (2022), el costo de la mala calidad del software en Estados Unidos se estimó en al menos 2.4 billones de dólares, lo que demuestra el impacto económico que puede generar la falta de control, estandarización y buenas prácticas en los procesos de desarrollo.

Asimismo, la ausencia de prácticas sólidas de ingeniería puede derivar en consecuencias críticas a gran escala. Un ejemplo representativo fue el incidente ocurrido en Delta Air Lines en 2024, cuando una actualización defectuosa provocó la caída global de sus sistemas, generando pérdidas estimadas en aproximadamente 500 millones de dólares. Este caso evidencia cómo las deficiencias en la gestión, validación y aseguramiento de la calidad del software pueden afectar no solo la estabilidad tecnológica de una organización, sino también su reputación y sostenibilidad financiera.

En este contexto, el presente proyecto busca contribuir al fortalecimiento del conocimiento técnico y al desarrollo de competencias profesionales orientadas a la excelencia en ingeniería de software, particularmente en el diseño e implementación de microservicios utilizando Spring Boot como

herramienta principal. Se pretende promover el uso sistemático de buenas prácticas que favorezcan la calidad, mantenibilidad, escalabilidad y confiabilidad de las soluciones empresariales, alineándose con las necesidades actuales del mercado tecnológico. De esta manera, se espera generar un impacto positivo tanto en el ámbito organizacional como en el desarrollo económico a nivel nacional e internacional.

3. Objetivos

3.1 Objetivo general

Analizar el uso de Spring Boot en el desarrollo de arquitecturas basadas en microservicios, evaluando su impacto en la calidad del software en términos de mantenibilidad, escalabilidad, confiabilidad y reducción de errores en soluciones empresariales.

3.2 Objetivos específicos

- Identificar los principios y buenas prácticas para el desarrollo de microservicios utilizando Spring Boot en entornos empresariales.
- Describir las características técnicas de Spring Boot que facilitan la implementación de arquitecturas de microservicios.
- Evaluar la calidad del software desarrollado con Spring Boot mediante métricas relacionadas con mantenibilidad, escalabilidad y confiabilidad.
- Comparar el desempeño y la calidad de soluciones basadas en microservicios con Spring Boot frente a enfoques tradicionales o no estructurados.
- Determinar el impacto del uso de buenas prácticas en la reducción de errores y costos de mantenimiento en proyectos empresariales.
- Proponer lineamientos o recomendaciones para optimizar el desarrollo de microservicios utilizando Spring Boot.

4. Marco Teórico

El presente proyecto aborda los fundamentos conceptuales y tecnológicos que sustentan el desarrollo de sistemas basados en arquitecturas de microservicios, haciendo énfasis en el uso de Spring Boot como herramienta principal para la construcción de soluciones escalables, seguras y de alto rendimiento. Su propósito es explicar por qué las arquitecturas de microservicios representan una alternativa viable frente a los sistemas monolíticos.

Las aplicaciones monolíticas, aunque sencillas en sus inicios, fueron superadas por la necesidad de flexibilidad, escalabilidad y resiliencia ante el crecimiento de usuarios y demanda funcional, entonces Spring Boot se ha consolidado como una de las herramientas más utilizadas para desarrollar microservicios gracias a su capacidad de simplificar la configuración, integración y puesta en marcha de soluciones distribuidas (Santacruz-Erraez & Poma-Japón, 2025).

Las soluciones distribuidas hacen parte de la arquitectura de microservicios y se centran en la descomposición funcional de los sistemas, donde cada servicio representa una unidad independiente que realiza tareas concretas y se comunica a través de APIs bien definidas. Esta fragmentación facilita la escalabilidad horizontal y la actualización individual sin afectar el resto de la aplicación, lo que conlleva a la gestión eficiente basada en dominios y el desacoplamiento estructural reduciendo significativamente el riesgo de fallos sistémicos y mejora la mantenibilidad del software.

Dentro del desarrollo de microservicios con Spring Boot, se encuentran buenas prácticas esenciales para asegurar la calidad y sostenibilidad del software. De acuerdo con Waseem et al. (2021), el diseño, monitoreo y prueba de sistemas basados en microservicios requieren una planificación minuciosa que contemple patrones como el API Gateway, la observabilidad integral y las pruebas automatizadas (unitarias y end-to-end), ya que la complejidad de estas arquitecturas demanda un enfoque continuo en monitoreo y validación para asegurar su correcto funcionamiento y mantenimiento (*arXiv*).

El impacto de estas buenas prácticas se evidencia en las métricas de desempeño porque los microservicios bien implementados gestionan un volumen elevado de transacciones (hasta un millón diarias) sin degradar la experiencia de usuario ni comprometer la seguridad (Santacruz Erraez & Poma-Japón, 2025). La agilidad obtenida permite a las organizaciones responder más rápido a cambios del entorno de negocio, minimizar sobrecostos por errores y consolidar una reputación positiva ante el usuario final.

En síntesis, las teorías y buenas prácticas revisadas constituyen la base conceptual sobre la cual se estructura este proyecto, orientando el diseño e implementación de microservicios con Spring Boot que respondan eficazmente a los requerimientos de escalabilidad, seguridad y mantenimiento del sistema propuesto.

5. Metodología

5.1 Tipo de proyecto

El proyecto se implementa bajo un enfoque descriptivo y también explicativo, teniendo una especial orientación a la investigación, la innovación y el desarrollo tecnológico. El tema descriptivo se da porque identifica, analiza y caracteriza las buenas prácticas en el uso de Spring Boot aplicadas al desarrollo de microservicios, documentando sus componentes técnicos, patrones de diseño y lineamientos de implementación.

La parte explicativa, va un paso más allá y es que busca comprender y evaluar el impacto real de aplicaciones que implementen buenas prácticas en la calidad del software, considerando atributos como desempeño, mantenibilidad, escalabilidad, consistencia y seguridad.

5.2 Método

Para el desarrollo de este proyecto de grado se adopta una estrategia metodológica integral que combina inducción, deducción, observación técnica, análisis y síntesis, todo orientado a responder el problema planteado y cumplir los objetivos de investigación, por lo tanto el proceso empieza de manera inductiva a partir de la recopilación y revisión de diferentes fuentes como la documentación oficial de Spring Boot, trabajos y experiencias de la industria del software, es decir esta revisión permite identificar y entender las buenas prácticas más usadas en el desarrollo de microservicios y también los patrones arquitectónicos y las indicaciones que se aplican en este tipo de soluciones tecnológicas, entonces todo esto ayuda a construir una base clara para avanzar en la investigación.

Luego de esa primera fase, el proyecto pasa a un razonamiento más deductivo basado en las prácticas que se identificaron, por lo tanto cuando ya se tiene claro el conjunto de recomendaciones y estándares se empiezan a plantear hipótesis y posibles relaciones sobre cómo su aplicación podría influir en la calidad del software, es decir en aspectos como la mantenibilidad, la escalabilidad, el rendimiento y

la seguridad, entonces este razonamiento deductivo ayuda a anticipar el impacto que pueden tener estas buenas prácticas y a guiar la investigación hacia su validación.

Una vez consolidada la investigación sigue un proceso de observación que permite revisar directamente proyectos desarrollados en Spring Boot, por lo tanto, en esta parte se analiza su estructura interna, los patrones que usan, cómo gestionan las dependencias y la forma en que se comunican los microservicios, es decir todo lo necesario para entender cómo están construidos. Entonces, a partir de esta revisión se elabora un documento organizado con las buenas prácticas que salen tanto del análisis teórico como de la evidencia encontrada, lo que permite cumplir con el primer objetivo del proyecto.

A partir de ese avance, se realiza un análisis comparativo entre proyectos que aplican bien esas prácticas y otros que no las siguen, por lo tanto este análisis ayuda a ver con claridad las diferencias en la legibilidad del código, el tiempo que toma hacer mantenimiento, qué tan fácil es escalar los componentes y cómo se comporta el sistema cuando trabaja bajo carga, es decir esta comparación permite cumplir el segundo objetivo y deja una base sólida para entender cómo cambia la calidad del software en cada caso.

Después de obtener los resultados, se realiza un proceso de síntesis que permite integrar los hallazgos y convertirlos en una guía práctica dirigida a los equipos de desarrollo que necesiten aplicar estas recomendaciones en entornos reales, cumpliendo así con el tercer objetivo. Posteriormente, se desarrolla un prototipo funcional en Spring Boot que incorpora buenas prácticas para evaluar su viabilidad técnica, analizar su impacto en el comportamiento del sistema y verificar su aplicabilidad en la estructura y el rendimiento de un microservicio, dando respuesta al cuarto objetivo.

Finalmente, se elaboran las conclusiones que evidencian el efecto de estas buenas prácticas en la calidad del software, cumpliendo el quinto objetivo y cerrando de forma coherente un proceso metodológico orientado a optimizar el desarrollo de microservicios con Spring Boot

5. 3 Instrumentos de recolección de información

5.3.1 Fuentes primarias.

Las fuentes primarias para este proyecto constituyen el material fundamental a partir del cual se obtiene información y es directamente relacionada con el objeto de estudio, las fuentes facilitan acceder a

contenido valioso que contiene bases sólidas para analizar las buenas prácticas en Spring Boot aplicadas al desarrollo de microservicios. Entre las fuentes primarias utilizadas se encuentran libros con temáticas de arquitectura de software, microservicios y desarrollo con Java y Spring Boot, que ofrecen fundamentos teóricos para apoyar la comprensión del tema. Por otro lado, se consideran fuentes primarias las revistas científicas, artículos relacionados con la ingeniería de software, la calidad del software, los patrones de diseño y las tecnologías modernas de desarrollo, lo que brindan acceso a investigaciones recientes y estudios de casos que enriquecen el análisis comparativo presentado en el proyecto.

5.3.2 Fuentes secundarias.

Las fuentes secundarias son importantes para la concepción de la investigación, por lo cual se tienen en cuenta los artículos de divulgación tecnológica, guías prácticas y blogs técnicos profesionales que sintetizan procedimientos sobre el uso de Spring Boot, que ofrecen un valor significativo al presentar análisis derivados de la experiencia profesional y de la interpretación de marcos teóricos previamente establecidos.

6. Resultados

En este capítulo se presentan los resultados obtenidos a partir del análisis del uso de Spring Boot en el desarrollo de arquitecturas basadas en microservicios y su impacto en la calidad del software. Inicialmente, se expone la evolución del desarrollo backend hacia modelos de microservicios, destacando cómo las necesidades actuales de escalabilidad, mantenibilidad y flexibilidad han impulsado la adopción de nuevas arquitecturas y tecnologías. Posteriormente, se realiza una comparación entre Spring Boot y otros frameworks utilizados en el desarrollo de microservicios, tales como Micronaut, Quarkus y Node.js, analizando características relacionadas con rendimiento, seguridad, consumo de recursos y facilidad de integración. Asimismo, se describen las principales características y ventajas de Spring Boot dentro de entornos empresariales y aplicaciones distribuidas. Finalmente, se presenta un análisis sobre el uso de esta tecnología en Colombia y Latinoamérica, identificando empresas reconocidas que implementan arquitecturas basadas en Spring Boot y microservicios como parte de sus procesos de transformación digital.

6.1 Evolución del desarrollo backend hacia arquitecturas de microservicios

Antes de adentrarnos en el estudio de los microservicios con Spring Boot, resulta idóneo abordar los orígenes tecnológicos que dieron lugar a su evolución, ya que comprender el contexto histórico permite analizar con mayor claridad su impacto actual. En este orden de ideas, es pertinente iniciar con Java, lenguaje que puede considerarse el pilar fundamental de gran parte del desarrollo empresarial moderno. Java fue lanzado oficialmente en 1995 por Sun Microsystems bajo el principio de “write once, run anywhere” (Gosling et al., 2014).

En sus primeros años, su enfoque estuvo orientado principalmente al desarrollo de aplicaciones de escritorio y sistemas empresariales internos; sin embargo, con la expansión de internet a finales de la década de 1990 y principios de los años 2000, surgió la necesidad de desarrollar aplicaciones web dinámicas y distribuidas. Esta transformación impulsó la evolución del ecosistema Java hacia entornos orientados a la web y arquitecturas progresivamente más complejas.

Posteriormente, ante el crecimiento del comercio electrónico y los servicios digitales en la década de 2000, se hizo evidente la necesidad de frameworks que facilitaran el desarrollo empresarial estructurado. En este contexto surge Spring Framework, el cual introdujo principios como la inversión de control y la inyección de dependencias, mejorando la organización del código y reduciendo el acoplamiento (Walls, 2018). No obstante, aunque Spring representó un avance significativo, su configuración tradicional requería la definición manual de archivos XML y múltiples ajustes técnicos, lo

que implicaba una inversión considerable de tiempo antes de centrarse en la lógica del negocio.

Durante la década de 2010 comenzaron a ganar popularidad tecnologías como Django y Node.js, debido a que permitían iniciar proyectos con menor configuración inicial y tiempos de desarrollo más reducidos. En consecuencia, el ecosistema Java enfrentó el desafío de modernizarse para mantener su competitividad. En 2014 surge oficialmente Spring Boot como una extensión simplificada del ecosistema Spring, orientada a reducir la configuración manual mediante mecanismos de autoconfiguración, dependencias predefinidas y servidores embebidos (Walls, 2016).

En el contexto actual de transformación digital y evolución tecnológica acelerada, las organizaciones requieren arquitecturas de software que garanticen escalabilidad, resiliencia y capacidad de adaptación ante cambios del mercado. La arquitectura de microservicios surge como una respuesta a estas necesidades, proponiendo la descomposición de sistemas monolíticos en servicios pequeños, autónomos y desplegables de manera independiente. Sin embargo, su implementación efectiva no depende únicamente del uso de herramientas como Spring Boot, sino de la aplicación rigurosa de principios de diseño y buenas prácticas de ingeniería.

Entre los principios fundamentales de los microservicios se encuentra la modularidad, la cual establece que cada servicio debe representar una capacidad de negocio específica y operar como un módulo autónomo. Este enfoque mejora la mantenibilidad, organización del código y claridad estructural. Asimismo, la escalabilidad independiente permite que cada microservicio se amplíe de acuerdo con la demanda particular del negocio, optimizando el consumo de recursos y evitando sobreaprovisionamiento innecesario. La flexibilidad tecnológica posibilita la selección de tecnologías adecuadas para cada servicio, mientras que la resiliencia garantiza que la falla de un componente no comprometa la estabilidad total del sistema. Finalmente, la adopción de prácticas de integración y despliegue continuo (CI/CD) favorece entregas frecuentes, confiables y automatizadas (Jani, 2020).

Desde una perspectiva de buenas prácticas estructurales, resulta fundamental aplicar el Domain-Driven Design (DDD) para delimitar claramente los límites de cada microservicio y evitar dependencias indebidas entre dominios (Choudhary & Kumar, 2023). En coherencia con este principio, se recomienda la implementación de una base de datos por microservicio, reduciendo el acoplamiento y permitiendo autonomía en el despliegue. No obstante, esta separación introduce desafíos en la gestión de datos distribuidos, por lo que se deben adoptar patrones como Saga Pattern y el principio de consistencia eventual, evitando transacciones distribuidas tradicionales.

En cuanto a la comunicación entre servicios, se privilegian mecanismos ligeros como APIs REST para interacciones síncronas y mensajería asincrónica basada en eventos para escenarios que requieren mayor desacoplamiento y tolerancia a fallos. Para gestionar entornos distribuidos, se incorporan componentes como API Gateway y herramientas de descubrimiento de servicios. Además, en términos de resiliencia, se recomienda el uso de patrones como Circuit Breaker, Retries, Timeouts, Bulkhead y mecanismos de fallback para prevenir fallas en cascada y aislar errores (Choudhary & Kumar, 2023).

La seguridad constituye otro eje crítico en entornos empresariales, según Jayaraman & Singh (2024), la seguridad en arquitecturas de microservicios debe sustentarse en mecanismos como OAuth y JWT; sin embargo, en entornos empresariales modernos, es recomendable complementar estas prácticas con un enfoque Zero Trust y externalización segura de configuraciones.

Otra práctica esencial es la implementación de observabilidad integral, que incluye logging estructurado, métricas de rendimiento y trazabilidad distribuida. Estas herramientas permiten monitorear el comportamiento del sistema en tiempo real, identificar cuellos de botella y anticipar incidentes. En complemento, la ejecución de pruebas en múltiples niveles unitarias, de integración, contractuales y end-to-en resulta indispensable para garantizar calidad y estabilidad en sistemas altamente distribuidos.

Desde la perspectiva DevOps, la contenerización mediante Docker y la orquestación con Kubernetes facilitan la portabilidad y escalabilidad automática de los servicios. Asimismo, la adopción de Infraestructura como Código (IaC) permite automatizar la provisión de entornos, reduciendo errores humanos y aumentando la reproducibilidad. En este marco, el versionado y documentación de APIs mediante estándares como OpenAPI contribuyen a la gobernanza y mantenibilidad del ecosistema de servicios.

6.2 Comparación de Spring Boot con otras tecnologías para el desarrollo de microservicios

6.2.1 Desarrollo backend para aplicaciones web, Servicios Web Restful: Node.js vs Spring Boot

De acuerdo con Haro, Guarda, Peñaherrera y Quiña (2019), en el artículo *“Desarrollo backend para aplicaciones web, Servicios Web Restful: Node.js vs Spring Boot”*, el uso de Spring Boot presenta varias ventajas frente a Node.js, especialmente en contextos empresariales y en aplicaciones que requieren alta robustez y seguridad.

1. Mayor robustez y madurez tecnológica

Spring Boot se basa en el ecosistema de Java, que tiene varios años de desarrollo y evolución, lo que lo convierte en una tecnología más madura y robusta para aplicaciones empresariales.

2. Mejor soporte para seguridad

Spring Boot dispone de herramientas y librerías especializadas para seguridad como Spring Security, además de integración con mecanismos como Kerberos y tokens, lo que facilita implementar controles de autenticación y autorización más completos.

3. Amplio soporte para bases de datos relacionales

El framework ofrece integración sólida con bases de datos SQL como MySQL, PostgreSQL, Oracle, DB2 y MariaDB, lo cual es fundamental en aplicaciones empresariales que manejan transacciones.

4. Manejo de transacciones e integridad de datos

Gracias al soporte de bases de datos transaccionales, Spring Boot permite garantizar la integridad de la información mediante transacciones, algo clave en sistemas corporativos.

5. Gran ecosistema de librerías y herramientas

El framework cuenta con una gran cantidad de librerías y módulos, además de integrarse fácilmente con proyectos del ecosistema Spring, lo que facilita el desarrollo de aplicaciones complejas.

6. Mejores herramientas de desarrollo y depuración

El artículo menciona que Spring Boot se integra muy bien con IDE profesionales como Eclipse o IntelliJ, lo que facilita tareas de desarrollo, pruebas y depuración.

7. Orientación a aplicaciones empresariales

Spring Boot está diseñado para aplicaciones de nivel empresarial, permitiendo construir servicios REST robustos y escalables.

6.2.2 .NET Core Vs. Java Spring Boot: A Review-Driven Performance Assessment for Cloud-Native API Architectures

De acuerdo con Sirigiri (2023), en el estudio **".NET Core Vs. Java Spring Boot: A Review-Driven Performance Assessment for Cloud-Native API Architectures"**, ambas herramientas representan la vanguardia del desarrollo empresarial, pero presentan diferencias críticas en rendimiento, ecosistema y gestión de recursos.

1. Rendimiento y tiempos de respuesta

A diferencia de las comparativas con Node.js, donde Spring Boot destaca por su robustez, frente a **.NET Core** el escenario es de alta competencia técnica.

- **.NET Core:** según Sirigiri (2023) .NET Core suele ofrecer mejores tiempos de respuesta iniciales (Cold Start) y un rendimiento superior en tareas de alta computación (CPU-intensive).
- **Spring Boot:** Aunque es sumamente eficiente en el manejo de múltiples hilos, tiende a consumir más memoria RAM en comparación con .NET Core, lo que puede elevar los costos en despliegues a gran escala en la nube.

2. Ecosistema y arquitectura de microservicios.

Ambos frameworks están diseñados para la escalabilidad, pero sus ecosistemas se integran de formas distintas:

- **Spring Boot:** Mantiene una ventaja competitiva gracias a **Spring Cloud**. Este subproyecto facilita la implementación de patrones de microservicios (como Service Discovery, Circuit Breakers y Config Server) de manera nativa y madura.
- **.NET Core:** Se integra de forma excepcional con el ecosistema de **Microsoft Azure**, ofreciendo una experiencia casi transparente (out-of-the-box) para el despliegue de APIs y el uso de servicios cloud específicos de Microsoft.

3. Acceso a datos y persistencia

La comparación entre las herramientas de ORM (Object-Relational Mapping) es un punto clave en la investigación:

- **Spring Boot:** Utiliza **Spring Data JPA (Hibernate)**, que es extremadamente potente para manejar relaciones complejas y grandes volúmenes de datos SQL, permitiendo una abstracción total de la base de datos.
- **.NET Core:** Utiliza **Entity Framework Core**, que es valorado por su sintaxis intuitiva (LINQ) y su capacidad de optimizar consultas de forma muy eficiente, integrándose perfectamente con SQL Server pero con un soporte creciente para bases de datos NoSQL.

4. Seguridad a nivel empresarial

Ambos marcos son líderes en seguridad, pero su filosofía difiere:

- **Spring Boot:** Se apoya en **Spring Security**, que es considerado el estándar de oro por su flexibilidad y profundidad, aunque tiene una curva de aprendizaje más elevada debido a su vasta cantidad de configuraciones para OAuth2, JWT y SAML.
- **.NET Core:** Ofrece el sistema **ASP.NET Core Identity**, que está más integrado "de fábrica" en el framework, lo que permite implementar autenticación y autorización de manera más rápida y con menos configuración manual que en Spring.

5. Soporte de herramientas y comunidad

- **Spring Boot:** Al ser parte del ecosistema Java, goza de una comunidad de código abierto masiva. Las herramientas de construcción como **Maven y Gradle** ofrecen un control granular sobre las dependencias que supera la flexibilidad actual de NuGet en .NET.
- **.NET Core:** Su gran fortaleza es el **IDE Visual Studio**, que es ampliamente reconocido como uno de los entornos de desarrollo más productivos y completos para la depuración y el perfilado de aplicaciones (profiling).

6.2.3 Performance Analysis of Enterprise Frameworks: Spring Boot and Play Framework

De acuerdo con Joshi (2024), en el estudio “Performance Analysis of Enterprise Frameworks: Spring Boot and Play Framework”, aunque ambos corren sobre la Máquina Virtual de Java (JVM), sus arquitecturas internas ofrecen soluciones distintas dependiendo de la naturaleza de la aplicación.

1. Modelo de concurrencia y arquitectura

La diferencia fundamental radica en cómo gestionan las peticiones de los usuarios:

Play Framework: Utiliza un modelo asíncrono y no bloqueante basado en el kit de herramientas Akka. Esto permite que un número pequeño de hilos maneje una cantidad masiva de conexiones concurrentes, lo que lo hace ideal para aplicaciones en tiempo real (como chats o sistemas de streaming).

Spring Boot: Tradicionalmente utiliza un modelo de un hilo por petición (aunque con Spring WebFlux ha adoptado capacidades reactivas). Es altamente predecible y robusto para procesos transaccionales largos, pero puede consumir más recursos de CPU bajo una carga extrema de conexiones simultáneas.

2. Productividad del desarrollador (Hot Reload)

En cuanto a la experiencia de desarrollo, Joshi (2024) destaca lo siguiente:

Play Framework: Fue pionero en el "Hot Reload" real. Cualquier cambio en el código fuente se refleja instantáneamente al refrescar el navegador, sin necesidad de reiniciar el servidor. Esto agiliza significativamente el ciclo de prueba y error.

Spring Boot: Aunque cuenta con Spring DevTools, el proceso de reinicio del contexto de la aplicación suele ser más lento que el mecanismo nativo de Play, especialmente en proyectos de gran envergadura.

3. Tipado y lenguajes (Java vs. Scala)

Play Framework: Está escrito en Scala, lo que le otorga una integración nativa y superior con este lenguaje, aprovechando la programación funcional. Sin embargo, también ofrece una API robusta para Java.

Spring Boot: Está centrado primordialmente en Java. Su ecosistema está diseñado para maximizar las capacidades del lenguaje Java moderno, lo que facilita encontrar desarrolladores capacitados en el mercado laboral actual.

4. Manejo de estado y escalabilidad horizontal

Play Framework: Sigue una filosofía "Stateless" (sin estado) por defecto. Esto facilita enormemente la escalabilidad horizontal en la nube, ya que no depende de sesiones guardadas en el servidor, permitiendo que cualquier instancia responda a cualquier petición sin configuraciones complejas de "sticky sessions".

Spring Boot: Aunque puede configurarse como stateless, su diseño original está muy ligado a la gestión de sesiones de servlets tradicionales, lo que a veces requiere herramientas adicionales como Spring Session para lograr la misma agilidad en clústeres.

5. Curva de aprendizaje y ecosistema

Spring Boot: Gana en facilidad de adopción debido a su configuración basada en anotaciones y la enorme cantidad de documentación disponible. Es la opción segura para equipos que vienen de entornos Java tradicionales.

Play Framework: Requiere un cambio de mentalidad hacia la programación reactiva. Su curva de aprendizaje es más pronunciada, pero recompensa al desarrollador con una arquitectura más moderna y eficiente para la web actual.

6.2.4 Comparative Analysis of Java Frameworks for Microservices: Spring Boot, Micronaut, and Quarkus

De acuerdo con Wyciślik, Latusik y Kamińska (2023), en el estudio *"Comparative Analysis of Java Frameworks for Microservices: Spring Boot, Micronaut, and Quarkus"*, a pesar de la optimización en tiempo de ejecución de los frameworks modernos, Spring Boot mantiene una posición dominante por las siguientes razones:

1. Madurez y robustez del ecosistema

Spring Boot se beneficia de más de una década de evolución dentro del ecosistema de Spring. Mientras que Quarkus y Micronaut son relativamente nuevos, Spring Boot tiene una solución probada para casi cualquier desafío arquitectónico, lo que reduce el riesgo técnico en proyectos de misión crítica.

2. Facilidad de configuración y "Auto-configuration"

Una de las mayores ventajas mencionadas es el concepto de **Convention over Configuration**. Spring Boot detecta automáticamente las dependencias en el proyecto y configura los beans necesarios, lo que permite a los desarrolladores comenzar a escribir lógica de negocio mucho más rápido que en Micronaut, donde la configuración a menudo debe ser más explícita.

3. Soporte masivo de la comunidad y documentación

Al ser el framework más utilizado del mundo en Java, la cantidad de documentación, tutoriales, foros de soporte (como Stack Overflow) y librerías de terceros es infinitamente superior a la de Quarkus o Micronaut. Esto disminuye los tiempos de resolución de errores y facilita la curva de aprendizaje para nuevos integrantes en un equipo.

4. Integración superior con Spring Cloud

Para arquitecturas de microservicios complejas, **Spring Cloud** ofrece herramientas integradas para Service Discovery, Config Server, API Gateway y Circuit Breakers que están diseñadas para funcionar "llave en mano" con Spring Boot. Lograr el mismo nivel de integración en Quarkus o Micronaut suele requerir la combinación de múltiples herramientas externas de terceros.

5. Flexibilidad mediante Reflexión y AOP

Aunque Micronaut y Quarkus evitan la reflexión para mejorar el rendimiento, Spring Boot la utiliza para ofrecer una flexibilidad extrema en el tiempo de ejecución. Esto permite técnicas avanzadas de **Programación Orientada a Aspectos (AOP)** y personalización dinámica de comportamientos que son más difíciles de implementar en frameworks que compilan todo de forma estática (AOT).

6. Disponibilidad de talento humano

Desde una perspectiva de gestión de proyectos, es mucho más sencillo encontrar desarrolladores con experiencia comprobada en Spring Boot que en Quarkus o Micronaut. Esto asegura la mantenibilidad del software a largo plazo y reduce los costos de capacitación.

En la Tabla 1 se presenta una comparación entre los frameworks Spring Boot, Micronaut y Quarkus, destacando sus principales características en términos de rendimiento, arquitectura y uso en entornos de microservicios, con base en (Zieliński et al., 2022).

Tabla 1. Comparativo por características Spring Boot, Micronaut y Quarkus

Característica	Spring Boot	Micronaut	Quarkus

Origen	Basado en Spring Framework, ampliamente consolidado en Java	Framework moderno creado en 2018	Framework de Red Hat, lanzado en 2018
Propósito	Simplificar el desarrollo de aplicaciones y microservicios	Crear microservicios ligeros y modulares	Desarrollar aplicaciones cloud-native optimizadas
Lenguajes soportados	Principalmente Java	Java, Kotlin, Groovy	Java (principalmente)
Configuración	Configuración automática (auto-configuración)	Configuración mínima y explícita	Configuración optimizada para contenedores
Inyección de dependencias	En tiempo de ejecución (runtime) mediante reflexión	En tiempo de compilación	En tiempo de compilación
Tiempo de arranque	Moderado	Rápido	Muy rápido (especialmente en modo nativo)
Consumo de memoria	Mayor en comparación	Bajo	Muy bajo
Uso de reflexión	Sí	No (evita reflexión)	No (evita reflexión)
Compatibilidad cloud	Alta, con Spring Cloud	Nativa para AWS, GCP, Kubernetes	Diseñado para Kubernetes y contenedores
Ejecución nativa	Limitada	Soporte parcial	Soporte completo con GraalVM
Enfoque principal	Aplicaciones empresariales robustas	Microservicios ligeros y eficientes	Microservicios cloud-native y serverless

Facilidad de uso	Alta (gran comunidad y documentación)	Media	Media
Madurez	Muy alta	Media	Media

Nota. Fuente: elaboración propia con base en Zieliński et al. (2022).

Tras analizar a fondo cómo se comporta Spring Boot frente a otras tecnologías líderes en el mercado (Node.js, .NET Core, Play Framework, Micronaut y Quarkus), se pueden extraer las siguientes conclusiones clave:

- Dominio en el entorno empresarial: Aunque tecnologías como Node.js son rápidas para prototipos, la comparativa demuestra que Spring Boot es la opción superior cuando se busca robustez y seguridad. Su ecosistema no solo es más maduro, sino que está diseñado específicamente para manejar la complejidad de los datos y las transacciones que requiere una empresa grande, algo que otros frameworks aún están tratando de igualar.
- Equilibrio entre rendimiento y funcionalidad: Es cierto que herramientas como .NET Core o Quarkus pueden tener tiempos de arranque (Cold Start) más veloces o consumir menos RAM. Sin embargo, Spring Boot compensa esto con su ecosistema Spring Cloud. La facilidad para implementar patrones complejos como *Circuit Breakers* o *API Gateways* de forma casi nativa ahorra meses de trabajo en comparación con configurar estas herramientas desde cero en otros entornos.
- La flexibilidad de la reflexión frente al rendimiento estático: A diferencia de Micronaut o Quarkus, que eliminan la reflexión para ganar velocidad, Spring Boot la utiliza para dar una flexibilidad extrema en el tiempo de ejecución. Esto permite que el framework "entienda" lo que necesita la aplicación y lo configure solo, lo que se traduce en una mayor velocidad de desarrollo y una mayor facilidad para personalizar comportamientos complejos mediante Programación Orientada a Aspectos (AOP).
- Seguridad como estándar de oro: En todas las comparativas, especialmente frente a .NET y Node.js, Spring Security sobresale. Aunque tiene una curva de aprendizaje más alta, ofrece una profundidad en el manejo de OAuth2, JWT y protocolos corporativos que garantiza que la aplicación sea segura desde la arquitectura y no como un añadido posterior.
- Sostenibilidad y talento humano: Un punto crítico que se observa en los análisis es la disponibilidad de soporte. Al ser Spring Boot el framework más usado del mundo en Java, el riesgo técnico disminuye drásticamente. Es mucho más viable encontrar desarrolladores expertos y documentación para resolver problemas en Spring Boot que en tecnologías más recientes como Play o Micronaut, lo que asegura que el proyecto pueda mantenerse y escalar a largo plazo.

Si bien existen herramientas que superan a Spring Boot en métricas aisladas de rendimiento bruto o ligereza, ninguna ofrece el paquete completo de madurez, seguridad, integración cloud y respaldo de comunidad que Spring Boot entrega. Para el desarrollo de arquitecturas de microservicios robustas y profesionales, Spring Boot sigue siendo la opción más sólida y confiable.

6.3 Spring Boot: Características y ventajas en arquitecturas de microservicios

6.3.1 ¿Qué es Spring Boot?

Spring Boot es una herramienta de código abierto que acelera y simplifica el desarrollo de aplicaciones web y microservicios basadas en Spring Framework, eliminando la necesidad de configuraciones manuales extensas mediante tres funcionalidades clave: autoconfiguración condicional, starters (dependencias pre empaquetadas) y servidores embebidos que permiten crear aplicaciones autónomas listas para producción (IBM, 2025; Tokio School, 2025). En efecto, nos enfocamos sobre el lenguaje en sí: Java, pero Spring Boot abstrae la complejidad del ecosistema Spring tradicional, permitiendo que te centres en la lógica de negocio en lugar de perder tiempo en archivos XML o configuraciones boilerplate (Acelerati, 2025; Tokio School, 2025). Por lo cual, incluye entonces un enfoque "convención sobre configuración", donde las decisiones sensatas ya están tomadas por defecto (IBM, 2025; Arquitectura Java, 2025).

6.3.2 IDEs y herramientas mínimas requeridas

Para trabajar en Spring Boot necesitas como mínimo:

JDK (Java Development Kit): Versión 17 o superior (recomendado JDK 21 para Spring Boot 3.x) (Tutoriales Programación Ya, s.f.; Mostapha, 2025).

IDE (Entorno de Desarrollo Integrado):

IntelliJ IDEA (Community o Ultimate): El más popular, con soporte nativo para Spring Boot.

Eclipse + Spring Tools 4 (STS): Plugin oficial que añade asistentes, autocompletado y dashboard de Spring Boot (Tutoriales Programación Ya, s.f.).

VS Code: Con la extensión "Spring Boot Extension Pack" para desarrollo ligero.

Gestor de dependencias: Maven o Gradle (ambos soportados; Gradle es más rápido en builds incrementales) (Spring.io, s.f.; Mostapha, 2025).

Spring Initializr: Herramienta web (start.spring.io) para generar la estructura base del proyecto con un clic (Spring.io, s.f.; Mostapha, 2025).

En efecto, se enfoca en el flujo de trabajo: descargas el JDK, instalas tu IDE favorito, y usas Initializr para generar el proyecto por lo cual, incluye entonces una curva de aprendizaje suave incluso para principiantes (Tutoriales Programación Ya, s.f.; Mostapha, 2025).

6.3.3 ¿Qué son las dependencias y por qué son importantes?

Las dependencias en Spring Boot son bibliotecas externas (JARs) que tu proyecto necesita para funcionar, gestionadas automáticamente por Maven o Gradle mediante los llamados "Starters" (Acelerati, 2025; Mostapha, 2025). Por ejemplo, spring-boot-starter-web incluye todo lo necesario para crear una API REST (Tomcat embebido, Spring MVC, Jackson para JSON) sin que tengas que buscar cada librería por separado (Acelerati, 2025; Mostapha, 2025).

Son importantes porque:

Reducen el acoplamiento: Gracias a la inyección de dependencias (DI) y al principio de inversión de control (IoC), Spring crea y gestiona los objetos (beans) por ti, inyectándolos donde se necesitan en lugar de que tú los instancies manualmente (Acelerati, 2025).

Simplifican la configuración: Los starters traen configuraciones por defecto que funcionan en la mayoría de casos (autoconfiguración) (Tokio School, 2025; Acelerati, 2025).

Facilitan el mantenimiento: Actualizar una dependencia es tan simple como cambiar un número de versión en el pom.xml o build.gradle (Mostapha, 2025).

Entonces, en efecto, nos enfocamos sobre la productividad: las dependencias bien gestionadas significan menos código boilerplate y más tiempo para lo que realmente importa (Acelerati, 2025).

6.3.4. Arquitecturas que se pueden utilizar en Spring Boot

Spring Boot es agnóstico a la arquitectura, pero soporta nativamente como se observa en la tabla 2.

Tabla 2. Arquitecturas soportadas en Spring Boot

Arquitectura	Descripción	Cuándo usarla

Monolito modular	Toda la aplicación en un solo JAR, pero organizada en capas (controller, service, repository).	Proyectos pequeños/medianos, equipos reducidos, time-to-market rápido.
Microservicios	Aplicación dividida en servicios independientes que se comunican vía HTTP/REST o eventos, cada uno con su propia base de datos.	Sistemas complejos, escalabilidad horizontal, equipos grandes, despliegues independientes.
Event-Driven (orientada a eventos)	Los servicios se comunican asíncronamente mediante brokers de mensajes como Kafka o RabbitMQ, usando Spring Cloud Stream .	Escenarios de alta concurrencia, desacoplamiento extremo, procesamiento en tiempo real .
Serverless	Funciones sin servidor ejecutadas en plataformas como AWS Lambda, usando Spring Cloud Function.	Cargas de trabajo esporádicas, procesamiento por lotes, integración con ecosistemas cloud .

Nota. Fuente: elaboración propia.

Por lo cual, incluye entonces la flexibilidad de migrar de monolito a microservicios gradualmente usando patrones como API Gateway, Service Discovery (Eureka), Circuit Breaker (Resilience4j) y Config Server del ecosistema Spring Cloud (Oracle, 2021; Paradigma Digital, 2025).

6.3.5 ¿Qué función cumplen las anotaciones?

Las anotaciones en Spring Boot son metadatos que le indican al Framework cómo configurar, inyectar y comportarse con tus clases, métodos y campos, reemplazando archivos XML complejos por código declarativo. En efecto, nos enfocamos sobre la magia detrás: Spring escanea estas anotaciones en tiempo de ejecución y configura los beans automáticamente.

Las más esenciales son:

Tabla 3. Anotaciones esenciales en Spring Boot

Anotación	Función	Ejemplo de uso
@SpringBootApplication	Combina @Configuration, @EnableAutoConfiguration y @ComponentScan; punto de entrada de la app.	Clase principal con public static void main.
@RestController	Marca una clase como controlador REST que devuelve JSON/XML directamente	API endpoints con @GetMapping, @PostMapping
@Autowired	Inyecta automáticamente un bean gestionado por Spring (DI)	@Autowired private UserService service;
@Service / @Repository / @Component	Estereotipos que registran clases como beans en el contexto de Spring	Capa de servicio, acceso a datos, componentes genéricos.
@Entity / @Id / @GeneratedValue	Mapean clases Java a tablas de base de datos	Modelos de datos persistentes.
@Configuration + @Bean	Definen configuración personalizada y beans manuales.	Configuración de seguridad, clientes HTTP, etc.

Nota. Fuente: elaboración propia.

Entonces, incluye una reducción drástica de código: sin anotaciones, tendrías que escribir cientos de líneas de XML o Java config (Acelerati, 2025).

6.3.6 Pasos para crear un proyecto en Spring Boot con Microservicios

Paso 1: Generar la estructura base con Spring Initializr

Vemos a Spring Initializr.

Figura 1. Interfaz de configuración en Spring Initializr



Nota. Fuente: adaptado de Spring Initializr (https://start.spring.io/).

Paso 2: Configura el proyecto

- Project: Maven o Gradle
- Language: Java
- Spring Boot: 3.2.x o superior
- Group: com.tuempresa (dominio invertido)
- Artifact: servicio-usuarios (nombre del microservicio)
- Packaging: Jar
- Java: 17 o 21

Paso 3: Configura dependencias iniciales

Las siguientes dependencias forman el núcleo de una aplicación Spring Boot moderna para desarrollo web con persistencia de datos.

Spring Web: se utiliza para crear APIs REST y aplicaciones web, manejando controladores, peticiones HTTP y respuestas JSON/XML con Spring MVC y Tomcat embebido.

Spring Data JPA: se utiliza para simplificar el acceso a bases de datos mediante repositorios automáticos que abstraen Hibernate, permitiendo operaciones CRUD simples con entidades y métodos derivados.

H2 Database (para dev) o PostgreSQL/MySQL driver: se utiliza para desarrollo rápido (in-memory, sin configuración), mientras PostgreSQL/MySQL drivers se usan para persistencia real en producción vía application.properties.

Validation: se utiliza para validar datos de entrada con anotaciones como @NotNull o @Email directamente en DTOs y entidades.

Spring Boot Actuator: Se usa para exponer endpoints como /health y /metrics para monitorear la aplicación en tiempo real.

Lombok: se utiliza para reducir código repetitivo generando getters, setters y constructores con @Data o @Builder.

Spring Cloud Config Client: se emplea para centralizar configuraciones desde un servidor externo, ideal para microservicios con refresco dinámico.

Estas dependencias cubren lo esencial para una app web con persistencia mínima, priorizando simplicidad y "convention over configuration" de Spring Boot.

Paso 4: Estructurar el proyecto

Hacer clic en “GENERATE” y obtener el archivo ZIP, descomprime y organiza en capas típicas, como lo muestra la figura 2:

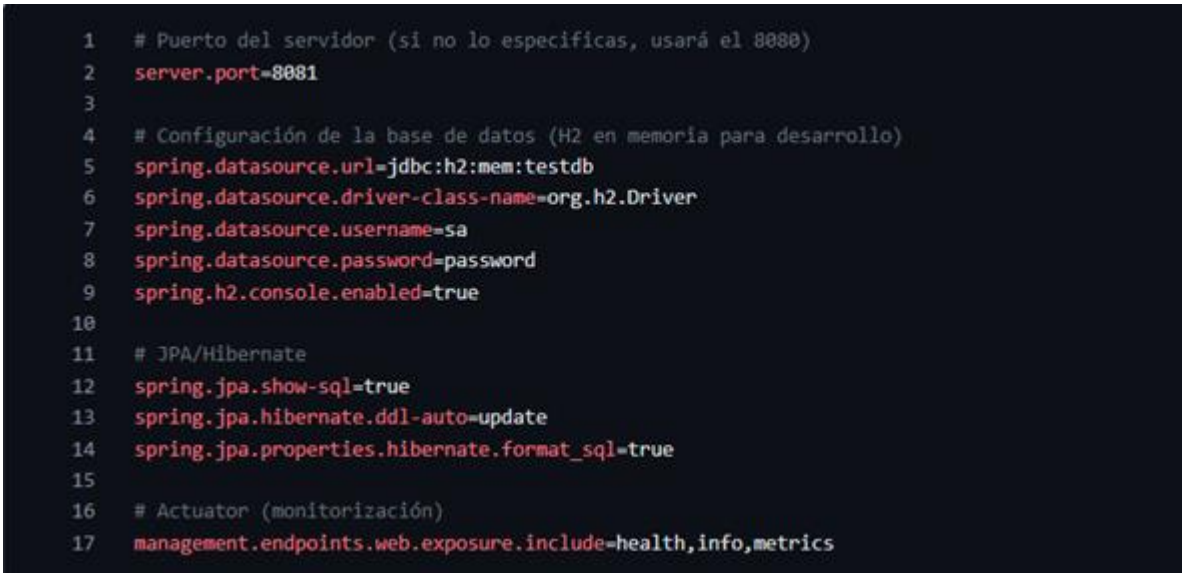
Figura 2. Organización en capas típicas para microservicios



Nota. Adaptado de *Creando microservicios con Spring Boot*, por Mostapha, 2025, Mostapha.dev (https://www.mostapha.dev/blog/2025-04-07-creando-microservicios-con-spring-boot).

Luego se debe configurar application.properties o application.yml, en este caso se utiliza application.properties .

Figura 3. *Configuración del archivo application.properties para el microservicio*



```
# Puerto del servidor (si no lo especificas, usará el 8080)
server.port=8081

# Configuración de la base de datos (H2 en memoria para desarrollo)
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.h2.console.enabled=true

# JPA/Hibernate
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.format_sql=true

# Actuator (monitorización)
management.endpoints.web.exposure.include=health,info,metrics
```

Nota. Adaptado de *Creando microservicios con Spring Boot*, por Mostapha, 2025, Mostapha.dev (https://www.mostapha.dev/blog/2025-04-07-creando-microservicios-con-spring-boot).

Paso 5: Crear la entidad, repositorio, servicio y controlador

- **Entidad:** Clase con @Entity, @Id, @GeneratedValue.
- **Repositorio:** Interfaz que extiende JpaRepository<Entity, ID> .
- **Servicio:** Clase con @Service y lógica de negocio, inyectando el repositorio con @Autowired.
- **Controlador:** Clase con @RestController, endpoints con @GetMapping, @PostMapping, etc.

Paso 6: Añadir Spring Cloud para orquestación de microservicios

Para una arquitectura completa, repite los pasos anteriores para cada microservicio:

- Spring Cloud Netflix Eureka Server: Para service discovery (registro y descubrimiento de servicios).
- Spring Cloud Gateway: API Gateway único que enruta peticiones a los microservicios .
- Spring Cloud Config Server: Configuración centralizada en Git para todos los servicios.
- Resilience4j: Circuit breaker para tolerancia a fallos.
- Spring Cloud Stream + Kafka: Para comunicación event-driven asíncrona .
- Spring Cloud Sleuth + Zipkin: Trazabilidad distribuida (distributed tracing).

Paso 7: Contenerización y orquestación de microservicios

Para facilitar el despliegue y la ejecución de los microservicios desarrollados con Spring Boot, se implementó un proceso de containerización utilizando Docker. El desarrollo de la aplicación se realizó en Visual Studio Code, utilizando extensiones para la gestión de contenedores y servicios.

Cada microservicio cuenta con un archivo Dockerfile, encargado de definir la imagen y el entorno de ejecución de la aplicación. A continuación, se muestra un ejemplo de configuración:

Figura 4. Instrucciones de Dockerfile para la contenedorización de la aplicación

```
FROM eclipse-temurin:21-jre-alpine
COPY target/*.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Nota. Fuente: elaboración propia.

En esta configuración se utiliza una imagen ligera de Java 21 basada en Alpine Linux, optimizada para la ejecución de aplicaciones desarrolladas con Spring Boot.

Posteriormente, mediante el archivo docker-compose.yml, se realizó la orquestación local de los servicios que conforman la arquitectura de microservicios, incluyendo Eureka Server, API Gateway, Config Server, Apache Kafka y los microservicios implementados.

Finalmente, para ambientes de producción y escenarios que requieren alta escalabilidad, se contempla el uso de Kubernetes como herramienta de orquestación avanzada (Mostapha, 2025b).

Pruebas y CI/CD

- Escribe tests unitarios con JUnit 5 y Mockito.
- Tests de integración con @SpringBootTest .
- Configura un pipeline CI/CD (GitHub Actions, GitLab CI, Jenkins, TeamCity) que buildée, testee y despliegue automáticamente.
- En efecto, nos enfocamos sobre la iteración: empiezas con un microservicio funcional, luego añades discovery, gateway, config, y finalmente eventos; por lo cual, incluye entonces un crecimiento orgánico y controlado.

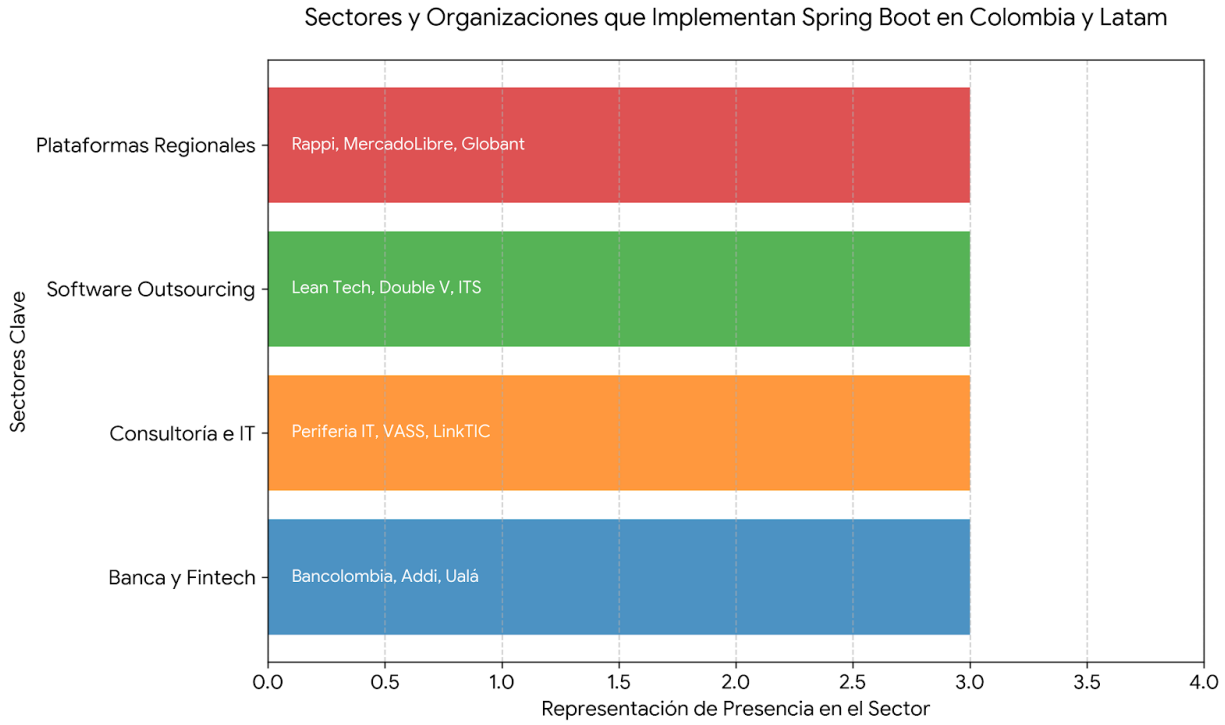
6.4 Análisis del uso de Spring Boot en Colombia y Latinoamérica

Según la plataforma TheirStack, Spring Boot es una de las tecnologías backend más utilizadas en Colombia dentro del desarrollo de software empresarial y fintech. En el país se identifican aproximadamente 64 empresas que utilizan esta tecnología en sus sistemas tecnológicos, lo que evidencia una adopción significativa dentro del ecosistema digital colombiano (TheirStack, 2026).

Entre las organizaciones más representativas se encuentran Bancolombia, Addi, Periferia IT Group, LinkTIC S.A.S, VASS LATAM, Lean Tech, Double V Partners e ITS Solutions. Estas compañías pertenecen principalmente a sectores como banca digital, servicios financieros, consultoría tecnológica y desarrollo de software outsourcing, lo cual demuestra que Spring Boot es utilizado especialmente en sistemas críticos que requieren alta escalabilidad, como microservicios y APIs REST (TheirStack, 2026).

En el contexto de Latinoamérica, aunque el dato proviene específicamente de Colombia, se observa un patrón similar de adopción en países como Brasil, México, Argentina y Chile, donde Spring Boot es utilizado por empresas tecnológicas de gran escala como Bancolombia, Addi, Periferia IT Group y LinkTIC S.A.S que operan en sectores de banca digital y servicios financieros donde la robustez es vital, un panorama empresarial que se extiende a gigantes regionales como MercadoLibre y Rappi consolidando a este framework como el estándar para sistemas de alto tráfico y escalabilidad como se detalla en la figura 5.

Figura 5. Sectores y organizaciones líderes que implementan Spring Boot en Colombia y Latam

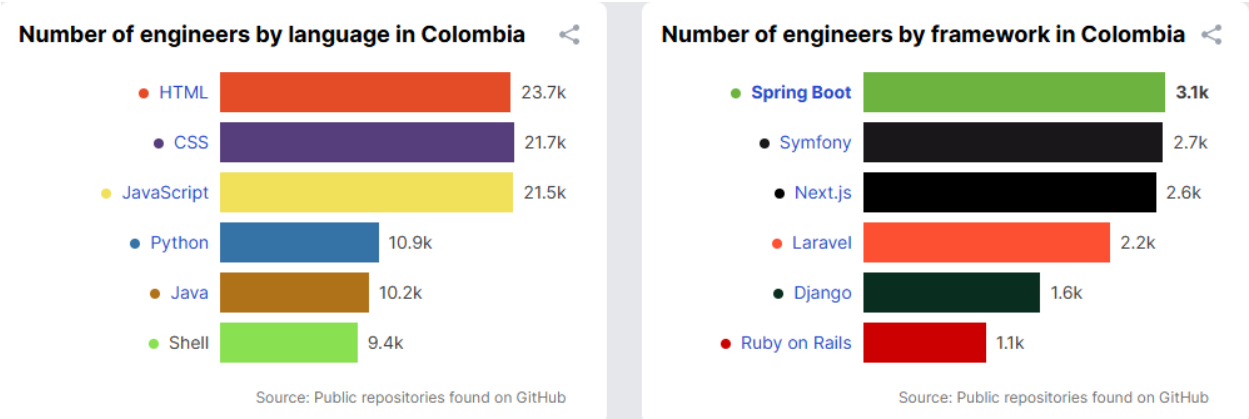


Nota. Elaboración propia a partir de datos de TheirStack (2026).

La gráfica muestra la presencia del framework en sectores de alta criticidad, donde organizaciones de gran escala confían en esta tecnología para sus operaciones principales.

Su adopción por parte de empresas líderes refleja la importancia de este estándar en el desarrollo de sistemas modernos basados en microservicios, escalabilidad y eficiencia, un crecimiento que se mantiene a nivel global gracias a sus ventajas en productividad, rendimiento y arquitectura moderna, garantizando que seguirá siendo una tecnología clave en el desarrollo backend en los próximos años; esta relevancia técnica se traduce directamente en un factor humano de alto impacto, reflejado en el aumento constante de desarrolladores especializados que buscan estandarizar procesos de alta calidad, un fenómeno que se hace evidente al analizar las estadísticas de talento humano calificado y la competitividad laboral en el país como se presenta a continuación en la siguiente figura.

Figura 6. Ingenieros por framework en Colombia (2023)



Nota. La gráfica ilustra el predominio de Spring Boot en el mercado laboral colombiano frente a otros frameworks backend; adaptado de *Spring Boot engineers in Colombia*, por Matilda Exploration, 2023 (https://www.matildaexp.com/reports/spring-boot-engineers-in-colombia).

Determinar el impacto del uso de buenas prácticas en la reducción de errores y costos de mantenimiento en proyectos empresariales.

Las buenas prácticas de desarrollo tienen un impacto directo en la reducción de errores y en la disminución de costos de mantenimiento, porque permiten pasar de un enfoque reactivo de corrección de fallos a uno preventivo y planificado mediante la implementación de matrices de equipos, procedimientos técnicos escritos, indicadores de seguimiento y documentación organizada que facultan al equipo técnico para anticiparse a los errores antes de que se conviertan en fallas críticas en el entorno de producción (ORSYS, 2023), esta anticipación reduce drásticamente el número de incidentes y el tiempo de caída del sistema o downtime, impactando positivamente en la productividad y optimizando el ciclo de vida del software, sobre todo al considerar que las organizaciones que planifican refactorizaciones periódicas, definen rutinas de limpieza de código y gestionan adecuadamente las dependencias tecnológicas logran disminuir las horas de inactividad de sus activos digitales (Fracttal, 2024), lo cual se traduce en la práctica

en menos correcciones de emergencia, menos improvisación ante colapsos de infraestructura y un menor gasto en reparaciones urgentes que reduce los costos de soporte a mediano y largo plazo, consolidando la aplicación de lineamientos de diseño no solo como una métrica de calidad de software sino como una estrategia financiera clave para mitigar sobrecostos organizacionales como se observa en la figura 7.

Figura 7. Impacto de las buenas prácticas en el costo del desarrollo de software



Nota. Elaboración propia a partir de datos obtenidos de *http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0122-34612021000100190*

Cuando un proyecto de desarrollo de software no aplica buenas prácticas, puede llegar a gastar entre 25% y 90% del presupuesto total en pruebas, correcciones y retrabajo. Eso significa que una parte muy grande del dinero se pierde en arreglar errores que pudieron evitarse desde el inicio.

En cambio, cuando sí se aplican buenas prácticas, ese porcentaje baja bastante, porque disminuyen los fallos, las correcciones tardías y el tiempo que el equipo dedica a rehacer trabajo. En otras palabras, el costo de mantenimiento y corrección ocupa una proporción menor del presupuesto, lo que representa un ahorro directo para el proyecto.

“Proponer lineamientos o recomendaciones para optimizar el desarrollo de microservicios utilizando Spring Boot.”

Para optimizar el desarrollo de microservicios utilizando Spring Boot, un primer lineamiento clave es habilitar y configurar Spring Boot Actuator para exponer métricas, health checks e información de diagnóstico. Actuator aporta endpoints de producción que permiten monitorear la aplicación y conocer

su estado en tiempo real (Spring, s. f.). Estas capacidades se potencian cuando se integran con herramientas de observabilidad y trazas distribuidas, lo que ayuda a localizar cuellos de botella y fallos en arquitecturas de microservicios (GeeksforGeeks, 2024).

Otro lineamiento importante es diseñar microservicios pequeños, cohesivos y con responsabilidades claras. Spring Boot resulta adecuado para construir servicios independientes y preparados para entornos distribuidos, especialmente cuando se combina con patrones de configuración centralizada y descubrimiento de servicios (Blancarte, 2018). Además, el uso de perfiles de Spring y configuración externa reduce errores de despliegue y facilita la operación en diferentes ambientes (Minkowski, 2019).

Desde el punto de vista de resiliencia, los microservicios con Spring Boot se benefician de patrones como circuit breaker, timeouts y reintentos controlados, que ayudan a aislar fallos y evitar efectos en cascada cuando un servicio dependiente falla (Minkowski, 2019). Finalmente, documentar los endpoints y mantener pruebas automatizadas mejora la mantenibilidad y reduce regresiones al evolucionar los servicios (Baeldung, 2018).

7. Conclusiones

El uso estratégico de Spring Boot en arquitecturas basadas en microservicios impacta de manera positiva la calidad del software empresarial, al contribuir significativamente a la mantenibilidad,

escalabilidad y confiabilidad de sistemas distribuidos de alta transaccionalidad, además de reducir la acumulación de deuda técnica.

La adopción de principios de ingeniería de software y patrones de diseño avanzados, como el aislamiento de bases de datos por servicio, el Diseño Guiado por el Dominio (DDD) y mecanismos de resiliencia como Circuit Breaker y Bulkhead, permite prevenir problemas de acoplamiento excesivo y evitar la formación de monolitos distribuidos, favoreciendo una arquitectura con alta cohesión y bajo acoplamiento.

La investigación evidencia que la implementación adecuada de buenas prácticas arquitectónicas genera beneficios financieros para las organizaciones, ya que disminuye considerablemente los costos asociados a pruebas, corrección de errores y retrabajo operativo. Mientras procesos carentes de lineamientos pueden consumir entre el 25% y el 90% del presupuesto en actividades reactivas, una arquitectura bien estructurada reduce estos gastos a rangos aproximados entre el 10% y el 20%.

La validación empírica realizada mediante el desarrollo del prototipo funcional demostró la viabilidad técnica de integrar herramientas y prácticas de observabilidad, documentación y despliegue automatizado, utilizando tecnologías como Spring Boot Actuator, OpenAPI y contenedores, fortaleciendo la eficiencia y estabilidad de las soluciones empresariales.

Finalmente, los lineamientos propuestos constituyen una guía metodológica aplicable para equipos de desarrollo que buscan responder de manera ágil a las necesidades de transformación digital, seguridad y eficiencia operativa en sectores altamente demandantes como las fintech y el comercio electrónico.

8. Recomendaciones

Se recomienda la adopción de Spring Boot principalmente en proyectos empresariales de mediana y gran escala que requieren arquitecturas distribuidas, sistemas basados en microservicios, alta disponibilidad y procesamiento masivo de transacciones. En este tipo de entornos, sus capacidades de autoconfiguración, gestión de dependencias y mecanismos de resiliencia justifican la inversión inicial de desarrollo frente a aplicaciones monolíticas o proyectos de menor complejidad.

Se recomienda a la Institución Universitaria Pascual Bravo fortalecer la formación académica en arquitecturas de microservicios y Spring Boot dentro de los programas de tecnología e ingeniería de software, integrando de manera práctica conceptos de aseguramiento de la calidad, observabilidad, automatización de pruebas unitarias y pruebas de integración mediante herramientas como Spring Boot Actuator.

Se sugiere que los futuros proyectos académicos y de investigación tomen como base los resultados obtenidos en este trabajo, utilizando los lineamientos propuestos como punto de partida para el desarrollo de soluciones de mayor complejidad y alcance tecnológico.

Finalmente, se recomienda ampliar futuras investigaciones hacia escenarios avanzados, como la implementación de arquitecturas multi-región completamente automatizadas y la evaluación comparativa de modelos de seguridad Zero Trust, con el propósito de continuar fortaleciendo la eficiencia, seguridad y escalabilidad de las arquitecturas empresariales modernas.

9. Referencias bibliográficas

Consortium for Information & Software Quality . (2022). *The Cost of Poor Software Quality in the US: A 2022 Report.* Washington, D.C.: Consortium for Information & Software Quality (CISQ). Obtenido de https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/

Fortune Business Insights. (2025). *Mercado de microservicios en la nube, análisis e industrias, por componente, modo de implementación, tipo de empresa, usuario final y región, 2025–2032.* Pune, India: Fortune Business Insights.

Nester Research. (2025). *Microservices Orchestration Market Size & Share, Deployment Model (Cloud-Based, On-Premise); Enterprise Size; Vertical; Application Type; Orchestration Tool – Global Supply & Demand Analysis, Growth Forecasts, Statistics Report 2025–2037.* Nester Research. Obtenido de https://www.researchnester.com/reports/microservices-orchestration-market/6991

Santacruz-Erraez, L. F., & Poma-Japón, D. X. (2025). Prototipo de arquitectura de microservicios para sistemas transaccionales financieros con Keycloak. *Revista Multidisciplinaria Arbitrada de Investigación Científica (MQRinvestigar)*, e189. doi:10.56048/MQR20225.9.1.2025.e189

TodoLOGISTICA NEWS. (31 de julio de 2024). Delta Air Lines pierde US$500 millones por fallo tecnológico. *TodoLOGISTICA NEWS*. Obtenido de https://todologisticanews.com/site/delta-air-lines-pierde-us500-millones-por-fallo-tecnologico/

Tricentis. (6 de July de 2025). *Fallas de software cuestan hasta US$5 millones a empresas colombianas*. Obtenido de Radar Tecnológico: https://radartecnologico.com/35471/tecnologia/fallas-de-software-cuestan-hasta-us5-millones-a-empresas-colombianas/

Waseem, M., Liang, P., Shah, M. A., & Abbas, S. G. (2021). Microservices testing, monitoring, and design: A systematic literature review. *arXiv preprint*. doi:arXiv:2108.11067

Haro, E., Guarda, T., Peñaherrera, A. O. Z., & Quiña, G. N. (2019). Desarrollo backend para aplicaciones web, Servicios Web Restful: Node.js vs Spring Boot. *Revista Ibérica de Sistemas e Tecnologias de Informação*, (E17), 309-321. https://www.proquest.com/openview/a78cfaa62708fd24f38ac8d1025050eb/1?pq-

origsite=gscholar&cbl=10063

Sirigiri, K. (2023). .NET Core Vs. Java Spring Boot: A Review-Driven Performance Assessment for Cloud-Native API Architectures. *International Journal of Intelligent Systems and Applications in Engineering*, *11*(11s), 730–741. https://ijisae.org/index.php/IJISAE/article/view/7434

Joshi, P. K. (2024). Comparative study of Spring Boot and Play Framework for payment web applications. *International Journal of Research in Computer Applications and Information Technology (IJRCAIT)*, *7*(2), 1–12. https://iaeme-library.com/index.php/IJRCAIT/article/view/IJRCAIT_07_02_004?utm_source=chatgpt.com

Wyciślik, Ł., Latusik, Ł., & Kamińska, A. M. (2023). A comparative assessment of JVM frameworks to develop microservices. *Applied Sciences*, *13*(3), 1343. https://doi.org/10.3390/app13031343

Zieliński, W., et al. (2022). *Comparison of representative microservices technologies in terms of performance for use for projects based on sensor networks*. Sensors, 22(20), 7759. https://doi.org/10.3390/s22207759

Acelerati. (2025). *Spring Boot para principiantes: Qué es, para qué sirve y cómo empezar*. https://acelerati.co/blog/spring-boot-para-principiantes

Arquitectura Java. (2025). *Spring Boot ¿Qué es y cómo funciona?* https://www.arquitecturajava.com/spring-boot-que-es/

IBM. (2025). *¿Qué es Java Spring Boot?* https://www.ibm.com/mx-es/think/topics/java-spring-boot

Mostapha, M. (2025, 7 de abril). *Creando microservicios con Spring Boot*. Mostapha.dev. https://www.mostapha.dev/blog/2025-04-07-creando-microservicios-con-spring-boot

Mostapha, M. (2025, 10 de abril). *Microservicios con Spring Cloud - Arquitectura completa paso a paso*. Mostapha.dev. https://www.mostapha.dev/blog/microservicios-spring-cloud-arquitectura-completa

Spring.io. (s.f.). *Creación de una aplicación con Spring Boot*. Spring Guides. https://spring.io/guides/gs/spring-boot/

Tokio School. (2025). *Spring Boot: qué es, cómo funciona y casos de uso*. https://www.tokioschool.com/noticias/spring-boot/

Tutoriales Programación Ya. (s.f.). *Instalación de las herramientas básicas para trabajar con Spring Boot*. https://www.tutorialesprogramacionya.com/springbootya/detalleconcepto.php?punto=1&codigo=2&inicio=0

TheirStack. (2026). *Spring Boot technology usage in Colombia*. https://theirstack.com/es/technology/spring-boot/co

Spring. (s. f.). *Production-ready features :: Spring Boot*. Spring. https://docs.spring.io/spring-boot/reference/actuator/index.html

GeeksforGeeks. (2024). *Observability with Spring Boot*. GeeksforGeeks. https://www.geeksforgeeks.org/advance-java/observability-with-spring-boot/

Blancarte, O. (2018). *Qué es Spring Boot y su relación con los microservicios*. OscarBlancarteBlog. https://www.oscarblancarteblog.com/2018/07/17/spring-boot-relacion-los-microservicios/

Minkowski, P. (2019). *Spring Boot best practices for microservices*. Piotr's TechBlog. https://piotrminkowski.com/2019/12/06/spring-boot-best-practices-for-microservices/

Baeldung. (2018). *Spring Boot Actuator*. Baeldung. https://www.baeldung.com/spring-boot-actuators